### 

### 🧵 “JavaScript is a ****synchronous single-threaded**** language”

✅ **Correct. But... there's more nuance.**

## 🧠 1. ****What This Actually Means****

| Term | Meaning |
| --- | --- |
| ✅ **Single-threaded** | JavaScript code (JS engine) runs in **one call stack**, one task at a time. |
| ✅ **Synchronous** | JavaScript executes **line by line** in **order**, **by default**. |

So:

console.log("A");

console.log("B");

console.log("C");

Prints:

A

B

C

This is **synchronous, single-threaded execution** — **the core behavior** of JS.

## 🧠 2. ****So How Does JS Handle Async Stuff?****

✅ Through **browser APIs** + **event loop**

➡️ The **JavaScript engine** is synchronous.  
➡️ But the **runtime environment (like browser or Node.js)** gives JS **async powers** (e.g. setTimeout, fetch, event listeners).

### Example:

console.log("Start");

setTimeout(() => console.log("Timeout done"), 1000);

console.log("End");

Prints:

Start

End

Timeout done

Why?

* setTimeout is sent to **browser Web API**
* The **callback is queued** for later via the **event loop**
* Main thread continues — no blocking!

✅ So even though **JS is single-threaded and synchronous**, it becomes **non-blocking and async** **with help from environment.**

## 🔄 Visual Flow:

1. JS enters code → top to bottom
2. Hits setTimeout → offloaded to browser
3. Continues rest of code (no wait!)
4. After timer → callback queued in **task queue**
5. **Event loop** checks: is call stack empty? Yes → push callback from queue!

## ✅ Summary

| Concept | Is it true? | Notes |
| --- | --- | --- |
| JS is synchronous by default | ✅ Yes | Executes top-down, line-by-line |
| JS is single-threaded | ✅ Yes | Only one call stack, no parallel execution |
| JS is asynchronous | ❌ Not by itself | But the runtime (browser/Node.js) provides async support |
| JS is non-blocking | ✅ With Promises, callbacks, etc., using event loop |  |

# JavaScript Execution Context

## ✅ What is Execution Context?

An **execution context** is the **environment in which JavaScript code is evaluated and executed**.

It contains everything the engine needs to run your code, like:

* variables
* functions
* the scope
* the this keyword

## 👨‍🏫 Every time you run JS code (line by line), JS creates an ****execution context**** for that block.

## 🧱 Components of Execution Context

As shown in your image, execution context has **2 major parts**:

### 1. ****Memory Component (Variable Environment)****

* Also called **“Creation Phase”**
* Stores **variables & function declarations**
* All variables initialized with undefined
* Functions are stored with full code (hoisting)

📦 Example:

var a = 10;

function fn() {

console.log("Hi");

}

➡️ In memory phase:

a: undefined

fn: [Function code]

### 2. ****Code Component (Thread of Execution)****

* Also called **“Execution Phase”**
* JS executes the code **line by line**
* Assigns real values to variables
* Runs function code when called

## 🔄 Two Phases of Execution Context

| Phase | Purpose |
| --- | --- |
| 🧠 Memory Phase | Allocates memory, sets up hoisting |
| ⚙️ Execution Phase | Executes the code top-to-bottom |

## 🪜 Types of Execution Context

### 1. ****Global Execution Context (GEC)**** 🌍

* Created automatically for every JS file
* Only **one GEC** exists at a time
* Variables/functions declared in the file go here

### 2. ****Function Execution Context (FEC)**** 🧩

* Created **every time a function is invoked**
* JS creates a **new execution context** for that function

### 3. ****Eval Execution Context**** (Rare)

* Used when eval() is called
* Not recommended, usually avoided

## 📚 Example: Dry Run

var x = 10;

function greet() {

var y = 20;

console.log(x + y);

}

greet();

### Step-by-Step Execution

🟢 **Global Execution Context is created**

* Memory:

x: undefined

greet: function

* Execution:

x = 10

greet() → triggers Function Execution Context

🟢 **Function Execution Context (for greet)**:

* Memory:

y: undefined

* Execution:

y = 20

console.log(10 + 20) → 30

## 📥 Call Stack

JavaScript uses a **Call Stack** to manage execution contexts:

| greet EC | ← pushed when greet() is called

| global EC | ← stays at bottom

Once greet() finishes:

| global EC | ← greet EC is popped off

## 🔄 How It All Ties Together

| Concept | Description |
| --- | --- |
| Execution Context | Environment where code runs |
| Memory Phase | Sets up variables/functions (hoisting) |
| Code Phase | Runs code line-by-line |
| Call Stack | Manages multiple function calls |

## ⚠️ Common Interview Confusion

* **Is JS synchronous or asynchronous?**

JS engine is **synchronous and single-threaded**  
Async behavior comes via **Web APIs + Event Loop**

* **What happens when you call a function?**

A new **execution context** is created and pushed to the **call stack**

* **Why variables are undefined before initialization?**

Due to **Memory Phase (hoisting)**

# ****JavaScript Event Loop – Explained Simply****

## 🧠 What is the Event Loop?

The event loop is a **mechanism** in JavaScript that helps handle **asynchronous operations** in a **single-threaded** environment — without blocking the main thread.

## 🚦 Key Components Involved

### 1. ****Call Stack****

* Executes your JavaScript code line by line.
* Follows **LIFO (Last In First Out)** structure.
* If a function is called, it’s **pushed onto the stack**, and removed once finished.

### 2. ****Web APIs (Browser or Node.js APIs)****

* Provided by the **browser** (e.g., setTimeout, DOM, fetch) or Node's libuv.
* These APIs run **outside JS engine**, often using threads (browser C++ threads or libuv in Node).
* When they finish their async task, they **register callbacks** into a queue.

### 3. ****Callback (Task) Queue****

* Holds callbacks from async operations (setTimeout, click, fetch).
* Waits for the **call stack to be empty**.

### 4. ****Event Loop****

* Constantly checks:

Is the call stack empty?  
If yes → takes the **first task from the queue** and pushes it onto the stack.

## 🧪 Example – Dry Run

console.log("Start");

setTimeout(() => {

console.log("Inside timeout");

}, 1000);

console.log("End");

### 💡 What happens:

1. "Start" → logged immediately
2. setTimeout() → Web API handles it, sets a timer
3. "End" → logged immediately
4. After 1 sec → callback is placed in the **callback queue**
5. Once call stack is empty → **event loop pushes** callback to stack
6. "Inside timeout" → gets logged

## 🧵 Is JavaScript Multithreaded?

❌ JavaScript itself is **single-threaded**  
✅ But the **browser (or Node)** provides multi-threaded **Web APIs** to handle async work.

## 🔄 Visual Model (like in Akshay Saini or Philip Roberts)

JS Engine

|

|--> Call Stack

|--> Web APIs (e.g., setTimeout, fetch)

|--> Callback Queue (aka Task Queue)

|--> Event Loop (the watcher)

## 🎯 What Makes Event Loop Powerful?

* JS can run **non-blocking async code**
* No need for threads like Java or C++
* Efficient for I/O, UI updates, networking, and timers

## ✅ Summary (Interview Level)

| Concept | Summary |
| --- | --- |
| JS Threading | Single-threaded engine |
| Web APIs | Provided by browser/Node to offload work |
| Call Stack | Where JS executes |
| Callback Queue | Holds async callbacks |
| Event Loop | Moves callbacks to stack once it's empty |